User-Definable Resource Bounds Analysis for Logic Programs

Jorge Navas[1], Edison Mera[2], Pedro López-García[3], and Manuel V. Hermenegildo[1,3]

[1] University of New Mexico, USA
[2] Complutense University of Madrid, Spain
[3] Technical University of Madrid, Spain

Abstract. We present a static analysis that infers both upper and lower bounds on the usage that a logic program makes of a set of user-definable resources. The inferred bounds will in general be functions of input data sizes. A *resource* in our approach is a quite general, user-defined notion which associates a basic cost function with elementary operations. The analysis then derives the related (upper- and lower-bound) resource usage functions for all predicates in the program. We also present an assertion language which is used to define both such resources and resource-related properties that the system can then check based on the results of the analysis. We have performed some preliminary experiments with some concrete resources such as execution steps, bytes sent or received by an application, number of files left open, number of accesses to a database, number of calls to a procedure, number of asserts/retracts, etc. Applications of our analysis include resource consumption verification and debugging (including for mobile code), resource control in parallel/distributed computing, and resource-oriented specialization.

1 Introduction

It is generally recognized that inferring information about the costs of computations can be useful for a variety of applications. These costs are usually related to execution steps and, sometimes, time or memory. We propose an analyzer which allows automatically inferring both upper and lower bounds on the usage that a logic program makes of *user-definable resources*. Examples of such user-definable resources are bits sent or received by an application over a socket, number of calls to a procedure, number of files left open, number of accesses to a database, number of licenses consumed, monetary units spent, disk space used, etc., as well as the more traditional execution steps, execution time, or memory. We expect the inference of this kind of information to be instrumental in a variety of applications, such as resource usage verification and debugging, certification of resource consumption in mobile code, resource/granularity control in parallel/distributed computing, or resource-oriented specialization.

In our approach a resource is a user-defined notion which associates a basic cost function with elementary operations in the base language and/or to some

predicates in libraries. In this sense, each *resource* is essentially a user-defined *counter*. The user gives a name (such as, e.g., `bits_received`) to the counter and then defines via assertions how each elementary operation in the program (e.g., unifications, calls to builtins, external calls, etc.) increments or decrements that counter. The use of resources obviously depends in practice on the sizes or values of certain inputs to programs or predicates. Thus, in the assertions describing elementary operations the counters may be incremented or decremented not only by constants but also by amounts that are *functions* of input data sizes or values. Correspondingly, the objective of our method is to statically derive from these elementary assertions and the program text *functions* that yield upper and lower bounds on the amount of those resources that each of the predicates in the program (and the program as a whole) will consume or provide. The input to these functions will also be the sizes or value ranges of the topmost input data to the program or predicate being analyzed.

As mentioned previously, most previous work is specific to the analysis of execution steps and, sometimes, time or memory. The ACE system [16] can automatically extract upper bounds on *execution steps* for a subset of functional programming. The system is based on program transformation. The original program is transformed into a step-counting version and then into a composition of a cost bound and a measure function. In [19] another automatic upper-bound analysis is presented based on an abstract interpretation of a step-counting version. The analysis measures both execution time and execution steps. However, size measures cannot automatically be inferred and the experimental section shows few details about the practicality of the analysis. In [8,7] a semi-automatic analysis is presented which infers upper-bounds on the number of execution steps. These bounds are functions on the sizes or value ranges of input data. This seminal work applies to a large class of logic programs and presents techniques in order to deal with the generation of multiple solutions via backtracking. The authors also show how other specific analyses could be developed, such as for, e.g., time or memory. This approach was later fully automated and extended to inferring upper- and lower-bounds on the number of *execution steps* (which is non-trivial because of the possibility of failure) in [9,12]. Our method builds on this work but generalizes it in order to deal with a much more general class of *user-defined* resources, allowing thus the coverage of an unlimited number of analyses within a single implementation. In [11] a method is presented for automatically extracting cost recurrences from first-order DML programs. The main feature is the use of dependent types to describe a size measure that abstracts from data to data size. In [17], and inspired by [3] and [15], a complexity analysis is presented for Horn clauses, also fully automating the necessary calculations. In [13] a method is presented for modeling problems such as memory management, lock primitive usage, etc., and a type-based method is proposed as solution to the inference problem. In [21] a cost model is presented for inferring cost equations for recursive, polymorphic, and higher-order functional programs. While it is claimed that the approach can be modified in order to infer a reduced set of resources such as execution time, execution steps, or memory, no details are given. Worst case execution time (WCET)

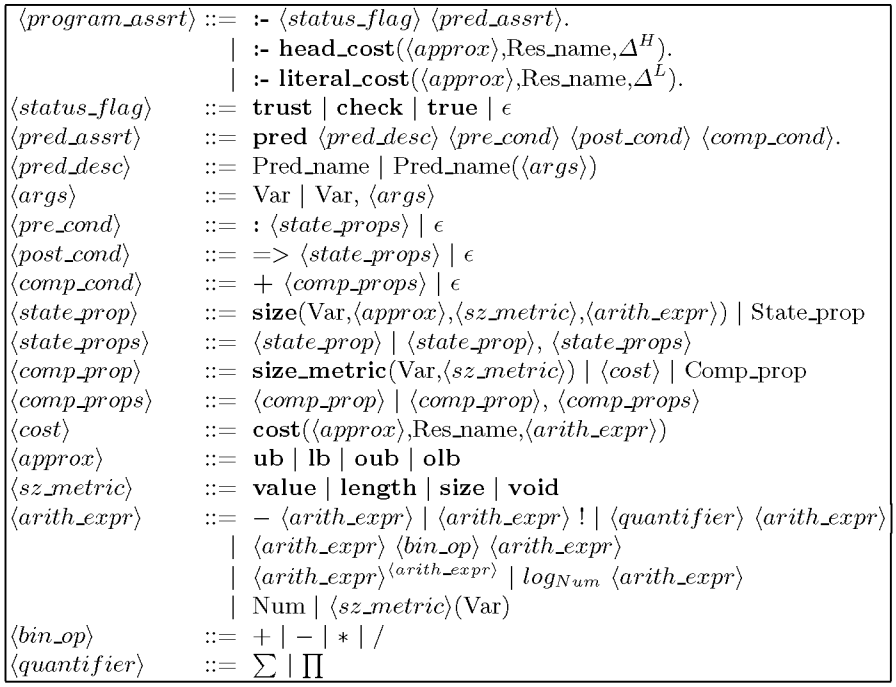
estimation has been studied for imperative languages and for different application domains (see, e.g., [20,4,10] and its references). However these and related methods again concentrate only on execution time. Also, they do not infer cost functions of input data sizes but rather absolute maximum execution times, and they generally require the manual annotation of loop iteration bounds. In [5] a method is presented for reserving resources before their actual use. However, the programmer (or program optimizer) needs to annotate the program with "acquire" and "consume" primitives, as well as provide loop invariants and function pre- and post-conditions. Interesting type-based related work has also been performed in the GRAIL system [1], also oriented towards resource analysis, but it has concentrated mainly on ensuring memory bounds.

In comparison with previous work our approach allows dealing with a class of resources which is open, in the ample sense that such resources are in fact defined by programmers using an assertion language, which we also consider itself an important contribution of our work. Another important contribution of our work (because its impact in the scalability and automation of the analysis) is that our approach allows defining the resource usage of external predicates, which can be used for modular composition. In addition, assertions also allow describing by hand the usage of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. In the following (Sect. 2) we first present the assertion language proposed. Sect. 3 then provides an overview of the approach, Sect. 4 shows how size relationships among program variables are determined, and Sect. 5 and 6 describe how the resource usage functions are inferred. We have implemented the proposed analysis and applied it to a series of example programs. The results are presented in Sect. 7. Finally, Sect. 8 summarizes our conclusions.

2 The Resource Assertion Language

We start by describing the assertion schema. This language is used for describing resources and providing other input to the resource analysis, and is also the language in which the resource analysis produces its output. This assertion language is used additionally to state resource-related specifications (which can then be proved or disproved based on the results of analysis following the scheme of [12], which allows finding bugs, verifying the program, etc.).

The rules for the assertion language grammar are listed in Fig. 1. In this grammar Var corresponds to variables written in the syntax for variables of the underlying logic programming language (i.e., normally non-empty strings of characters which start with a capital letter or underscore). Similarly, Num is any valid number and $Pred\_name$ any valid name for a predicate in the underlying programming language (normally non-empty strings of characters which start with a lower-case letter or are quoted). $State\_prop$ corresponds to other *state properties* (such as modes and types), and $Comp\_prop$ stands for any other valid *computational property* (see [12] and its references).



$\langle program\_assrt \rangle$	::=	**:-** $\langle status\_flag \rangle$ $\langle pred\_assrt \rangle$.
	|	**:- head_cost**($\langle approx \rangle$,Res_name,Δ^H).
	|	**:- literal_cost**($\langle approx \rangle$,Res_name,Δ^L).
$\langle status\_flag \rangle$	::=	**trust** | **check** | **true** | ϵ
$\langle pred\_assrt \rangle$	::=	**pred** $\langle pred\_desc \rangle$ $\langle pre\_cond \rangle$ $\langle post\_cond \rangle$ $\langle comp\_cond \rangle$.
$\langle pred\_desc \rangle$	::=	Pred_name | Pred_name($\langle args \rangle$)
$\langle args \rangle$	::=	Var | Var, $\langle args \rangle$
$\langle pre\_cond \rangle$	::=	: $\langle state\_props \rangle$ | ϵ
$\langle post\_cond \rangle$	::=	=> $\langle state\_props \rangle$ | ϵ
$\langle comp\_cond \rangle$	::=	+ $\langle comp\_props \rangle$ | ϵ
$\langle state\_prop \rangle$	::=	**size**(Var,$\langle approx \rangle$,$\langle sz\_metric \rangle$,$\langle arith\_expr \rangle$) | State_prop
$\langle state\_props \rangle$	::=	$\langle state\_prop \rangle$ | $\langle state\_prop \rangle$, $\langle state\_props \rangle$
$\langle comp\_prop \rangle$	::=	**size_metric**(Var,$\langle sz\_metric \rangle$) | $\langle cost \rangle$ | Comp_prop
$\langle comp\_props \rangle$	::=	$\langle comp\_prop \rangle$ | $\langle comp\_prop \rangle$, $\langle comp\_props \rangle$
$\langle cost \rangle$	::=	**cost**($\langle approx \rangle$,Res_name,$\langle arith\_expr \rangle$)
$\langle approx \rangle$	::=	**ub** | **lb** | **oub** | **olb**
$\langle sz\_metric \rangle$	::=	**value** | **length** | **size** | **void**
$\langle arith\_expr \rangle$	::=	$-$ $\langle arith\_expr \rangle$ | $\langle arith\_expr \rangle$! | $\langle quantifier \rangle$ $\langle arith\_expr \rangle$
	|	$\langle arith\_expr \rangle$ $\langle bin\_op \rangle$ $\langle arith\_expr \rangle$
	|	$\langle arith\_expr \rangle^{\langle arith\_expr \rangle}$ | log_{Num} $\langle arith\_expr \rangle$
	|	Num | $\langle sz\_metric \rangle$(Var)
$\langle bin\_op \rangle$	::=	$+$ | $-$ | $*$ | $/$
$\langle quantifier \rangle$	::=	$\sum$ | $\prod$

Fig. 1. Syntax of the Resource Assertion Language

Predicates can be annotated with zero or more **pred** assertions, which state properties of the execution states when the predicate is called (*pre_cond*), properties of the execution states when the predicate terminates execution (*post_cond*), or properties of the whole computation of the predicate, rather than the input-output behavior (*comp_cond*, which herein will be used only for resource-related properties). For brevity, the $\langle state\_props \rangle$ fields can also be written using "star notation" (see the examples). In addition, there may be a set of global **head_cost** and **literal_cost** declarations (one for each resource and approximation direction). The *Res_name* fields determine which resource the assertion refers to. These *Res_name*s are user-provided identifiers which give a name to each particular resource that needs to be tracked. Resources do not need to be declared in any other way –the set of resources that the system is aware of is simply the set of such names that appear in assertions which are in the scope. The $\langle approx \rangle$ fields state whether $\langle arith\_expr \rangle$ is providing an upper bound or a lower bound (with **oub** meaning it is a "big O" expression, i.e., with only the order information, and **olb** meaning it is an Ω asymptotic lower bound).

The first and most fundamental use of assertions in our context is to describe how the execution of some predicates increments or decrements the usage of the resources (counters) defined in the program. The purpose of analysis is then to infer the resource usage of all predicates in the program. The **head_cost**($\langle approx \rangle$, Res_name, Δ^H) declarations are used to describe how predicates in general *update*

the value for those resources that are applicable only to predicate heads (such as counting the number of arguments passed or total execution steps –see Section 5). The definition of Δ^H : *cl_head* $\rightarrow$ *arith_expr* is provided by means a user-defined (or imported) predicate, written in the source language, and which will be called by the analyzer when the clause head is analyzed. This code gets loaded into the compiler in a similar way to, e.g., macro expansion code. The **literal_cost**($\langle approx \rangle$, Res_name, Δ^L) declarations describe how predicate bodies *update* the value of certain resources which are applicable only to body literals (such as, for example, number of unifications). In this case, Δ^L : *body_lit* $\rightarrow$ *arith_expr* is also user- (or library-)provided code which will be executed when the body literals of different predicates are analyzed. The **cost**($\langle approx \rangle$, Res_name, $\langle arith\_expr \rangle$) comp-type properties are included in **pred** assertions and used to provide the actual resource usage functions for each builtin or external (e.g., defined in another language) predicate used in the program. Such assertions have **trust** status (meaning that the analysis will assume this value [12]). As mentioned previously, the aim of the analysis is to derive functions that describe the resource usage (as well as argument size relations) for the rest of the predicates in the program. Note however that it is also possible to provide **pred** assertions for some of those predicates and this can also be used to guide the analysis. In particular, the analysis will compute the most precise expression between the resource usage function provided by the assertion ($\langle cost \rangle$) and the resource usage function inferred by analysis. Additionally, size metric (**size_metric**(Var,$\langle sz\_metric \rangle$)) information can be provided by users if needed (but note that in practice size metrics can often be derived automatically from the inferred types).

Assertions can also be used, via the *pre_cond* and *post_cond* fields, to declare relationships between the data sizes of the inputs and outputs of predicates, which are needed by our analysis, as will be described later. These assertions are also used to label predicate arguments as input or output, as well as to provide types or size (**size**(Var,$\langle approx \rangle$,$\langle sz\_metric \rangle$,$\langle arith\_expr \rangle$)) information. In the same way as with the $\langle cost \rangle$ properties, for user-defined predicates these other assertions can be provided by the user or inferred by analysis. Again, analysis will compute the most precise of the two.

Example 1. The following example shows how a program can be annotated using our assertion language in order to perform resource analysis. Consider a client application in Fig 2 that sends a data buffer through a socket[1] and receives another (possibly transformed) data buffer. Assume that we would like to obtain an upper bound on the number of bits received by the application –a *resource* that we will call `bits_received`.

As we will see, the approach needs to know for each argument in the program the metric and whether it is input or output in order to perform properly the size and resource usage analyses described in Sect. 4 and 5. Input/output and metric information can be required by the language (typed language), given by the user (via assertions), or, as in our implementation, inferred automatically via analysis.

[1] Note that the types of the socket operations must be given to the analysis by other analyses or by user-provided assertions.

```
:- pred main(Opts, IBuf, OBuf)
   : list(gnd) * list(byte) * var.
main([Host,Port],IBuf,OBuf) :-
   connect_socket(Host,Port,Stream),
   exchange_buffer(IBuf,Stream,OBuf),
   close(Stream).
exchange_buffer([],_,[]).
exchange_buffer([B|Bs],Id,[B0|Bs0]) :-
   exchange_byte(B,Id,B0),
   exchange_buffer(Bs,Id,Bs0).
:- trust pred
   connect_socket(Host,Port,Stream)
   :  atm * num * var
   => atm * num * atm
   +  cost(ub,bits_received,0).
```

```
:- trust pred close(Stream)
   :  atm => atm
   +  cost(ub,bits_received,0).
:- trust pred exchange_byte(B,Id,B0)
   :  byte * num * var
   => byte * num *
     {byte(B0), size(B0,ub,bytes,1)}
   +  cost(ub,bits_received,8).

:- head_cost(ub,bits_received,
             head_bits_received).
head_bits_received(_,0).
:- literal_cost(ub,bits_received,
             literal_bits_received).
literal_bits_received(_,0).
```

Fig. 2. A simple client application

3 Overview of the Approach

Our basic approach is as follows. Given a predicate call p, let $\Phi(p, r, n)$ denote the units of resource r consumed or produced during the computation of p for an input of size n. An expression $\mathtt{RU}_{pred}(p, ap, r, n)$ is determined (at compile-time) that approximates $\Phi(p, r, n)$ with approximation ap. Typical size metrics are the actual value of a number, the length of a list or array, the size (number of nodes and fields) of a data structure, etc. We will refer to such $\mathtt{RU}_{pred}(p, ap, r, n)$ expressions as *resource usage bound functions*. Certain program information (such as, for example, input/output modes and size metrics for predicate arguments) is first automatically inferred by other (abstract interpretation-based) analyzers and then provided as input to the size and resource analysis (the techniques involved in inferring this information are beyond the scope of this paper —see, e.g., [12] and its references for some examples). Based on this information, our analysis first finds bounds on the size of input arguments to the calls in the body of the predicate being analyzed, relative to the sizes of the input arguments to this predicate using the inferred metrics. The size of an output argument in a predicate call depends in general on the size of the input arguments in that call. For this reason, for each output argument we infer an expression which yields its size as a function of the input data sizes. To this end, and using the input-output argument information, data dependency graphs are used to set up difference equations whose solution yields size relationships between input and output arguments of predicate calls. This information regarding argument sizes is then used to set up another set of difference equations whose solution provides bound functions on resource usage. Both the size and resource usage difference equations must be solved by a difference equation solver. Although the operation of such solvers is beyond the scope of the paper our implementation does provide a table-based solver which covers a reasonable set of difference equations such as first-order and higher-order linear difference equations in one variable with

constant and polynomial coefficients,[2] divide and conquer difference equations, etc. In addition, the system allows the use of external solvers (such as, e.g. [2], Mathematica, Matlab, etc.). Note also that, since we are computing upper/lower bounds, it suffices to compute upper/lower bounds on the solution of a set of difference equations, rather than an exact solution. This allows obtaining an *approximate* closed form when the exact solution is not possible.

4 Size Analysis

We will give the intuition behind the data dependency-based method for inferring bounds on the sizes of output arguments in the head of a predicate as a function of the sizes of input arguments to the predicate. Besides this, as a result of the size analysis, we have bounds on the size of each input argument to body literals in a clause as a function of the size of the input arguments to the head of that clause. The size of the input arguments to body literals will be used later to infer functions which give bounds on the resource usage of body literals in terms of the sizes of the input arguments to the head. We adopt the approach of [8,7] for the inference of upper bounds on argument sizes and [9] for lower bounds. For the sake of brevity, we will only consider the inference of upper bounds in this paper, and refer the reader to [9] for the inference of lower bounds.

Various metrics are used for the "size" of an argument (e.g., *value*, *length*, *size*,[3]...). For the sake of brevity, we will only consider the *length* metric in this paper, and refer the reader to [8,7,9] for other metrics.

We define the $\texttt{size}(\langle sz\_metric\rangle, t)$ operation which returns the size of a term t under the metric $\langle sz\_metric\rangle$:

$$\texttt{size}(length, t) = \begin{cases} 0 & \text{if } t = [\,] \text{ (the empty list)} \\ 1 + \texttt{size}(length, T) & \text{if } t = [H|T] \\ \bot & \text{otherwise.} \end{cases}$$

Thus, $\texttt{size}(length, [X, Y]) = 2$, and $\texttt{size}(length, [X|Y]) = \bot$.

We also define the $\texttt{diff}(\langle sz\_metric\rangle, t_1, t_2)$ operation, which returns (an upper bound on) the size difference between two terms t_1 and t_2 under the metric $\langle sz\_metric\rangle$.

$$\texttt{diff}(length, t_1, t_2) = \begin{cases} 0 & \text{if } t_1 \equiv t_2 \\ \texttt{diff}(length, t, t_2) - 1 & \text{if } t_1 = [\_|t] \text{ for some term t} \\ \bot & \text{otherwise.} \end{cases}$$

Thus, $\texttt{diff}(length, [a, b|T], T) = -2$.

A directed acyclic graph called *argument dependency graph* is used to represent the data dependency between argument positions in a clause body (and between them and those in the clause head). Each node in the graph denotes an argument position. There is an edge from a node n_1 to a node n_2 if the variable

[2] Note that it is always possible to reduce a system of linear difference equations to a single linear difference equation in one variable.

[3] Metric *void* in an argument means that analysis does not need to infer its size.

bindings generated by n_1 are used to construct the term occurring at n_2. The node n_1 is said to be a *predecessor* of the node n_2, and n_2 a *successor* of n_1.

Using the `size` and `diff` functions and the argument dependency graph we can set up size relations for expressing the size of each argument position in terms of the sizes of its predecessors. Let $\mathtt{sz}(a)$ denote the size of the term occurring at an argument position a, and $@a$ the term occurring at an argument position a. For the sake of simplicity, we will omit the argument $\langle sz\_metric \rangle$ in the `size` and `diff` functions in the rest of the paper.

- *Output arguments.* Let $l_1, \ldots, l_n$ denote the input argument positions of the literal L, and let $\Psi_p^b : \mathcal{N}_{\perp,\infty}^n \mapsto \mathcal{N}_{\perp,\infty}$ be a function that represents the size of the b-th (output) argument position of the predicate p of literal L in terms of the size of its input positions, where $\mathcal{N}_\perp$ denotes the set of natural numbers augmented with the special symbols $\perp$ (denoting "undefined"), and ∞.
  Assume that a is an output argument position in a clause. Then the following size relation is set up:
  $$\mathtt{sz}(a) \leq \Psi_p^a(\mathtt{sz}(l_1), \ldots, \mathtt{sz}(l_n))$$
  If L is recursive, then $\Psi_p^a(\mathtt{sz}(l_1), \ldots, \mathtt{sz}(l_n))$ is a symbolic expression. However, if L is non-recursive then the function Ψ_p^a has been recursively computed, and thus we replace $\Psi_p^a(\mathtt{sz}(l_1), \ldots, \mathtt{sz}(l_n))$ by the (explicit) expression resulting from the application of the function Ψ_p^a to $\mathtt{sz}(l_1), \ldots, \mathtt{sz}(l_n)$.
- *Input arguments.* Assume that a is an input argument position in a body literal. Let $\mathtt{predecessors}(a)$ be the set of predecessors of a in the argument dependency graph. We have the following possibilities:
  1. Compute $\mathtt{size}(@a)$. If $\mathtt{size}(@a) \neq \perp$ then set up the size relation: $\mathtt{sz}(a) \leq \mathtt{size}(@a)$.
  2. Otherwise, if $\exists r \in \mathtt{predecessors}(a)$ such that the size metrics corresponding to r and a are the same and $d = \mathtt{diff}(r, a) \neq \perp$, then $\mathtt{sz}(a) \leq \mathtt{sz}(r) + d$.
  3. Otherwise, if $\mathtt{size}(@a)$ can be expanded using the definition of the `size` function, then expand $\mathtt{size}(@a)$ one step and recursively compute $\mathtt{size}(t_i)$ for the appropriate subterms t_i of $@a$. If each of these recursive size computations have a defined result, then use them to compute the size relation for $\mathtt{size}(@a)$. Otherwise, $\mathtt{sz}(a) = \perp$.

Size relations can be propagated to transform a size relation corresponding to an input argument in a body literal or an output argument in the clause head into a function in terms of the sizes of the input arguments of the head. The basic idea here is to repeatedly substitute size relations for body literals into size relations for head arguments. This is the purpose of the normalization algorithm described in [7]. However, for recursive clauses, we need to solve the symbolic expression due to recursive literals into an explicit function first.

Example 2. Consider again the program described in Fig. 2. Here and in the rest of the paper we will denote by *pred_name* the name of a predicate, and by $pred\_name_i^j$ the i-th argument position in the j-th literal with predicate

name *pred_name* in the body of a clause. If there is only one body literal with predicate name *pred_name* in the body of a clause then we omit the superscript j and write simply $pred\_name_i$. Let $head_i$ be the i-th argument position in the clause head. To simplify notation, we will represent `exchange_buffer/3` and `exchange_byte/3` with *ex_buf* and *ex_byt* respectively. Consider the recursive clause of `exchange_buffer/3`. First, the system sets up the size relation for the input/output arguments of the body literals:

$$\begin{array}{rcl}
\mathtt{sz}(ex_byt_1) & \leq & \mathtt{size}(B) = \mathtt{sz}(arg(1, head_1)) \\
\mathtt{sz}(ex_byt_2) & \leq & \mathtt{size}(Id) = \mathtt{sz}(head_2) + \mathtt{diff}(Id, Id) = \mathtt{sz}(head_2) \\
\mathtt{sz}(ex_byt_3) & \leq & \Psi^3_{ex_byt}(\mathtt{sz}(ex_byt_1), \mathtt{sz}(ex_byt_2)) = 1 \\
\mathtt{sz}(ex_buf_1) & \leq & \mathtt{size}(Bs) = \mathtt{sz}(head_1) + \mathtt{diff}([B|Bs], Bs) = \mathtt{sz}(head_1) - 1 \\
\mathtt{sz}(ex_buf_2) & \leq & \mathtt{size}(Id) = \mathtt{sz}(head_2) + \mathtt{diff}(Id, Id) = \mathtt{sz}(head_2) \\
\mathtt{sz}(ex_buf_3) & \leq & \Psi^3_{ex_buf}(\mathtt{sz}(ex_buf_1), \mathtt{sz}(ex_buf_2))
\end{array}$$

Then, system sets up the size relation for the output arguments of the head:

$$\begin{array}{rcl}
\mathtt{sz}(head_3) & \leq & \mathtt{size}([B0|Bs0]) = \mathtt{size}(Bs0) + 1 = \\
& & \mathtt{sz}(ex_buf_3) + \mathtt{diff}(Bs0, Bs0) + 1 = \mathtt{sz}(ex_buf_3) + 1
\end{array}$$

The normalization algorithm is applied to the previous size relation and gives:

$$\begin{array}{rcl}
\mathtt{sz}(head_3) & \leq & \Psi^3_{ex_buf}(\mathtt{sz}(ex_buf_1), \mathtt{sz}(ex_buf_2)) + 1 \\
& \leq & \Psi^3_{ex_buf}(\mathtt{sz}(head_1) - 1, \mathtt{sz}(head_2)) + 1
\end{array}$$

Thus, the system establishes the difference equation for the output argument ($head_3$) in the head (since it belongs to a recursive predicate). Then, it obtains the boundary condition $\Psi^3_{ex\_buf}(0, y) = 0$ from the non-recursive clause, and using it, it obtains a closed form function by calling the difference equation solver (variables x and y represent $\mathtt{sz}(head_1)$ and $\mathtt{sz}(head_2)$ respectively):

$$\begin{array}{rcl}
\Psi^3_{ex_buf}(0, y) & = & 0 \\
\Psi^3_{ex_buf}(x, y) & = & \Psi^3_{ex_buf}(x - 1, y) + 1
\end{array} \qquad \Rightarrow \Psi^3_{ex_buf}(x, y) = \quad x$$

5 Resource Usage Analysis

In order to infer the resource usage functions all predicates in the program are processed in a single traversal of the call graph in reverse topological order. Consider such a predicate `p` defined by clauses $\mathtt{C}_1, ..., \mathtt{C}_m$. Assume that $\overline{n}$ is a tuple such that each element corresponds to the size of an input argument position to predicate `p`. Then, the resource usage (expressed in units of resource r with approximation ap) of a call to `p`, for an input of size $\overline{n}$, can be expressed as:

$$\mathtt{RU}_{\mathtt{pred}}(p, ap, r, \overline{n}) = \bigodot(ap)_{1 \leq i \leq m} \{\mathtt{RU}_{\mathtt{clause}}(C_i, p, ap, r, \overline{n})\} \tag{1}$$

where $\bigodot(ap)$ is a function that takes an approximation identifier ap and returns a function which applies over all $\mathtt{RU}_{\mathtt{clause}}(C_i, p, ap, r, \overline{n})$, for $1 \leq i \leq m$. For example, if ap is the identifier for approximation "upper bound" (ub), then a possible conservative definition for $\bigodot(ap)$ is the $\sum$ function. In this case, and since the number of solutions generated by a predicate that will be demanded is generally not known in advance, a conservative upper bound on the computational cost of

a predicate is obtained by assuming that all solutions are needed, and that all clauses are executed (thus the cost of the predicate is assumed to be the sum of the costs of all of its clauses). However, it is straightforward to take mutual exclusion into account (information which is inferred by CiaoPP [14,12] and is available to our analysis) to obtain a more precise estimate of the cost of a predicate, using the maximum of the costs of mutually exclusive groups of clauses. If ap is the identifier for approximation "lower bounds" (lb), then $\bigodot(ap)$ is the min function.

Let us see now how to compute the resource usage of clauses. Consider a clause $\mathtt{C}$ of predicate p of the form $\mathtt{H} :- \mathtt{L_1}, ..., \mathtt{L_k}$ where $\mathtt{L_j}$, $1 \leq j \leq k$, is a literal (either a predicate call, or an external or builtin predicate), and $\mathtt{H}$ is the clause head. Assume that $\psi_j(\overline{n})$ is a tuple with the sizes of all the input arguments to literal $\mathtt{L_j}$, given as functions of the sizes of the input arguments to the clause head. Note that these $\psi_j(\overline{n})$ size relations have previously been computed during size analysis for all input arguments to literals in the bodies of all clauses.

Then, $\mathtt{RU}_{\mathtt{clause}}(C, ap, r, \overline{n})$, the resource usage (expressed in units of resource r with approximation ap) of clause $\mathtt{C}$ of predicate p, is $\mathtt{RU}_{\mathtt{clause}}(C, ap, r, \overline{n}) = closed\_form(\mathtt{RU}(p, ap, r, \overline{n}))$. That is, it is expressed as the solved form function of the following expression (which, in general, for recursive clauses yields a difference equation):

$$\begin{array}{l} \mathtt{RU}(p, ap, r, \overline{n}) = \delta(ap, r)(head(C)) + \\ \sum_{j=1}^{lim(ap,C)} (\prod_{l \prec j} \mathtt{Sols}_{\mathtt{L}_l}(\overline{n}_l))(\beta(ap, r)(L_j) + \mathtt{RU}_{\mathtt{lit}}(L_j, ap, r, \psi_j(\overline{n}))) \end{array} \quad (2)$$

where $lim(ap, C)$ is a function that takes an approximation identifier ap and a clause C and returns the index of a literal in the clause body. For example, if ap is the identifier for approximation "upper bound" (ub), then $lim(ap, C) = k$ (the index of the last body literal). If ap is the identifier for approximation "lower bounds" (lb), then $lim(ap, C)$ is the index for the rightmost body literal that is guaranteed not to fail. $\delta(ap, r)$ is a function that takes an approximation identifier ap and a resource identifier r and returns a function $\Delta^H : cl\_head \rightarrow arith\_expr$ which takes a clause head and returns an arithmetic resource usage expression $< arith\_expr >$ as defined in Section 2. Thus, $\delta(ap, r)(head(C))$ represents $\Delta^H(head(C))$. On the other hand, $\beta(ap, r)$ is a function that takes an approximation identifier ap and a resource identifier r and returns a function $\Delta^L : body\_lit \rightarrow arith\_expr$ which takes a body literal and returns an arithmetic resource usage expression $< arith\_expr >$ as defined in Section 2. In this case, $\beta(ap, r)(L_j)$ represents $\Delta^L(L_j)$. $\mathtt{Sols}_{\mathtt{L}_l}$ is the number of solutions that literal $\mathtt{L}_l$ can generate, where $l \prec j$ denotes that $\mathtt{L}_l$ precedes $\mathtt{L}_j$ in the literal dependency graph for the clause. Section 6 illustrates different definitions of the functions $\delta(ap, r)$ and $\beta(ap, r)$ in order to infer different resources.
$\mathtt{RU}_{\mathtt{lit}}(\mathtt{L_j}, ap, r, \psi_j(\overline{n}))$ is:

- If $\mathtt{L_j}$ is recursive (i.e., calls a predicate $\mathtt{q}$ which is in the strongly-connected component of the call graph being analyzed), then $\mathtt{RU}_{\mathtt{lit}}(\mathtt{L_j}, ap, r, \psi_j(\overline{n}))$ is replaced by a symbolic expression $\mathtt{RU}(q, ap, r, \psi_j(\overline{n}))$.

– If $\mathtt{L_j}$ is not recursive, assume that it is a call to $\mathtt{q}$ (where $\mathtt{q}$ can be either a predicate call, or an external or builtin predicate), then $\mathtt{q}$ has been already analyzed, i.e., the (closed form) resource usage function for $\mathtt{q}$ has been recursively computed as Φ' and $\mathtt{RU_{lit}}(\mathtt{L_j}, ap, r, \overline{n})$ can be expressed explicitly in terms of the function Φ', and it is thus replaced with $\Phi'(\psi_j(\overline{n}))$.

Note that in both cases, if there is a resource usage assertion for $\mathtt{q}$, $cost(ap, r, \Phi)$, then $\mathtt{RU_{lit}}(\mathtt{L_j}, ap, r, \psi_j(\overline{n}))$ is replaced by the most precise (greatest lower bound if *upper bounds* or least upper bound if *lower bounds*) of a) the arithmetic resource usage expression (in closed form) $\Phi(\psi_j(\overline{n}))$ or b) its (closed form) resource usage function inferred previously by the analysis (provided they are not incompatible, in which case an error is flagged).

It can be proved by induction on the number of literals in the body of clause $\mathtt{C}$ that:

1. If clause $\mathtt{C}$ is not recursive, then, expression (2) results in a closed form function of the sizes of the input argument positions in the clause head;
2. If clause $\mathtt{C}$ is simply recursive, then, expression (2) results in a difference equation in terms of the sizes of the input argument positions in the clause head;
3. If clause $\mathtt{C}$ is mutually recursive, then expression (2) results in a difference equation which is part of a system of equations for mutually recursive clauses in terms of the sizes of the input argument positions in the clause head.

If these difference equations can be solved (including approximating the solution in the direction of ap) then $\mathtt{RU}(p, ap, r, \overline{n})$ can be expressed in a closed form, which is a function of the sizes of the input argument positions in the head of predicate $\mathtt{p}$ (and hence $\mathtt{RU_{clause}}(C, ap, r, \overline{n}) = closed\_form(\mathtt{RU}(p, ap, r, \overline{n}))$). Thus, after the strongly-connected component to which p belongs in the call graph has been analyzed, we have that expression (1) results in a closed form function of the sizes of the input argument positions in the clause head.

Note that our analysis is parameterized by the functions $\delta(ap, r)$ and $\beta(ap, r)$ whose definitions can be given by means of assertions of type **head_cost** and **literal_cost** respectively, as given in Figure 1. These functions make our analysis parametric w.r.t. resources (such as execution steps, bytes sent by an application/socket, number of calls to a procedure, etc.).

6 Defining the Parameters (Functions) of the Analysis

In this section we explain and illustrate with examples how the functions that make our resource analysis parametric, namely, δ (which includes the definition of Δ^H), and β (which includes the definition of Δ^L) are written in practice in our system. For brevity, we assume that we are interested in computing upper bounds on the different resources.

Assume for example that the resource we want to measure is (an upper bound on) the number of resolution steps performed by a program. Then we can define

$\delta(ub, steps) = delta\_one$, and define $delta\_one(H) = 1$ for all H. This is achieved by providing the following `head_cost` assertion and definition of the `delta_one` predicate:

```
:- head_cost(ub, steps, delta_one).
delta_one(_, 1).
```

In order to simplify the process of defining interesting and useful Δ^H and Δ^L functions, our implementation provides a library with predicates that perform (syntactic) operations on clauses, such as, for example, getting the number of arguments in a clause head or body literal, accessing an argument of a clause head or body literal, getting the main functor and arity of a term in a certain position, etc. In this context it is important to remember that the different Δ^H and Δ^L function definitions perform syntactic matching on the program text.

Assume now that the resource we want to measure is the number of argument passings that occur during clause head matching in a program (as an approximation to the number of unifications performed by the program). Then we can define $\delta(ub, num\_args) = delta\_num\_args$, and define $delta\_num\_args(H) = arity(H)$. This is achieved by the following code:

```
:- head_cost(ub, num_args, delta_num_args).
delta_num_args(H, N) :- functor(H, _, N).
```

As another example, if we are interested in decomposing arbitrary unifications performed while unifying a clause head with the literal being solved into simpler steps, we can define a resource num_unifs, define $\delta(ub, num\_unifs) = delta\_num\_unifs$, and define $delta\_num\_unifs(H) =$ *The number of function symbols, constants, and variables in* H.

If, in addition to the number of unifications performed while unifying a clause head, we are also interested in the cost of term creation for the literals in the body of clauses, we can define a resource $terms\_created$, and define a β function $\beta(ub, terms\_created) = beta\_terms\_created$, where $beta\_terms\_created(L) =$ *The number of function symbols, and constants in body literal* L. Note that in this case we define $\delta(ub, terms\_created) = delta\_terms\_created$, and define $delta\_terms\_created(H) = 0$ for all H.

Example 3. Consider the same program defined in Fig. 2 and the size relations computed in *Example* 2. We now show the corresponding resource usage equations for each clause for the resource *bits_received* (denoted by *bits* for brevity) inferred automatically by our system. Although the functions $\delta(ap, r)(H)$ and $\beta(ap, r)(L)$ take as arguments a clause head H and a body literal L respectively, in our examples we will only write the predicate name of H and L for the sake of brevity. Again, to simplify notation we will represent `exchange_buffer/3`, `exchange_byte/3` and `connect_socket/3` with *ex_buf* and *ex_byt* and *con_sock* respectively. Since the program is analyzed in a single traversal of the call graph in reverse topological order, the system starts by analyzing the predicate `exchange_buffer/3`. Note that the resource usage for external predicates (whose code is not available) `connect_socket/3`, `exchange_byte/3` and `close/1` is

already given by "trust" assertions which express that: $\texttt{RU}(con\_sock, \mathbf{ub}, bits, \_) = 0$, $\texttt{RU}(ex\_byt, \mathbf{ub}, bits, \_) = 8$ and $\texttt{RU}(close, \mathbf{ub}, bits, \_) = 0$.

For the recursive clause of `exchange_buffer/3`, the system sets up the following difference equation, where n represents the length of the first argument to this predicate (note that the system infers the "length" size metric for this argument and that $n > 0$):

$$\begin{aligned}\texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle n \rangle) = \quad & \delta(ub, bits)(ex\_buf) + \beta(ub, bits)(ex\_byt) + \\ & \texttt{RU}(ex\_byt, \mathbf{ub}, bits, \_) + \beta(ub, bits)(ex\_buf) + \\ & \texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle n-1 \rangle) = \\ & 0 + 0 + 8 + 0 + \texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle n-1 \rangle)\end{aligned}$$

For the non-recursive clause of `exchange_buffer/3` the system infers: $\texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle 0 \rangle) = 0$ which can be used as boundary condition for solving the previous difference equation, yielding the following closed form resource usage function: $\texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle n \rangle) = 8 \times n$.

Now, the `main/2` predicate is analyzed, and the system sets up the following expression for its only clause (where k is the length of the input buffer, i.e., the second argument to this predicate):

$$\begin{aligned}\texttt{RU}(main, \mathbf{ub}, bits, \langle k \rangle) = \quad & \delta(ub, bits)(main) + \\ & \beta(ub, bits)(con\_sock) + \texttt{RU}(con\_sock, \mathbf{ub}, bits, \_) + \\ & \beta(ub, bits)(ex\_buf) + \texttt{RU}(ex\_buf, \mathbf{ub}, bits, \langle k \rangle) \\ & \beta(ub, bits)(close) + \texttt{RU}(close, \mathbf{ub}, bits, \_) \\ = \quad & 0 + 0 + 0 + 0 + 8 \times k + 0 + 0 = 8 \times k\end{aligned}$$

7 Experimental Results

To study the feasibility of the approach we have completed a prototype implementation of the analyzer. It is written in the Ciao language and uses a number of modules and facilities from CiaoPP, the Ciao preprocessor (including difference equation processing). We have also written a Ciao language extension (a "package" in Ciao terminology) which when loaded into a module allows writing the resource-related assertions and declarations proposed herein.[4] We have then used this prototype to analyze a set of representative benchmarks which include definitions of resources using this language and used the system to infer the resource usage bound functions.

The results from the analysis of these benchmarks are shown in Table 1. For brevity, we report only results for upper-bounds analysis. The column labeled **Resource** shows the actual resource for which bounds are being inferred by the analysis for a given benchmark. While any of the resources defined in a given benchmark could then be used in any of the others we show only the results for the most natural or interesting resource for each one of them. We have tried to use a relatively wide range of resources: number of bytes sent by

[4] The system also supports adding resource assertions specifying expected resource usages which the implemented analyzer will then verify or falsify using the results of the implemented analysis.

Table 1. Accuracy and efficiency in milliseconds of the analysis

Program	Resource	Usage Function	Metrics	Time
`client`	"bits received"	$\lambda x.8 \cdot x$	length	186
`color_map`	"unifications"	39066	size	176
`copy_files`	"files left open"	$\lambda x.x$	length	180
`eight_queen`	"queens movements"	19173961	length	304
`eval_polynom`	"FPU usage"	$\lambda x.2.5x$	length	44
`fib`	"arithmetic operations"	$\lambda x.2.17 \cdot 1.61^x + 0.82 \cdot (-0.61)^x - 3$	value	116
`grammar`	"phrases"	24	length/size	227
`hanoi`	"disk movements"	$\lambda x.2^x - 1$	value	100
`insert_stores`	"accesses Stores" "insertions Stores"	$\lambda n, m.n + k$ $\lambda n, m.n$	length	292
`perm`	"WAM instructions"	$\lambda x.(\sum_{i=1}^{x} 18 \cdot x!) + (\sum_{i=1}^{x} 14 \cdot \frac{x!}{i}) + 4 \cdot x!$	length	98
`power_set`	"output elements"	$\lambda x.\frac{1}{2} \cdot 2^{x+1}$	length	119
`qsort`	"lists parallelized"	$\lambda x.4 \cdot 2^x - 2x - 4$	length	144
`send_files`	"bytes read"	$\lambda x, y.x \cdot y$	length / size	179
`subst_exp`	"replacements"	$\lambda x, y.2xy + 2y$	size / length	153
`zebra`	"resolution steps"	30232844295713061	size	292

an application, number of calls to a particular predicate, robot arm movements, number of files left open in a kernel code, number of accesses to a database, etc. The column **Usage Function** shows the actual resource usage function (which depends on the size of the input arguments) inferred by the analysis, given as a lambda term. The column **Metrics** shows the size metric used for the relevant arguments. Finally, the column labeled **Time** shows the resource analysis times in milliseconds, on a medium-loaded Pentium IV Xeon 2.0Ghz with two processors, 4Gb of RAM memory, running Fedora Core 5.0. Note that these times do not include other analyses such as types, modes, etc.

8 Conclusions

We have presented a static analysis that infers upper and lower bounds on the usage that a logic program makes of a quite general notion of user-definable resources. The inferred bounds are in general functions of input data sizes. We have also presented the assertion language which is used to define such resources. The analysis then derives the related (upper- and lower-bound) resource usage functions for all predicates in the program. Our preliminary experimental results are encouraging because they show that interesting resource bound functions can be obtained automatically and in reasonable time, at least for our benchmarks. While clearly further work is needed to assess scalability we are cautiously hopeful in the sense that our approach allows defining via assertions the resource usage of external predicates, which can then be used for modular composition.

These includes also predicates for which the code is not available or which are written in a programming language that is not supported by the analyzer. In addition, assertions also allow describing by hand the usage of any predicate for which the automatic analysis infers a value that is not accurate enough, and this can be used to prevent inaccuracies in the automatic inference from propagating. Our expectation is that the automatic analysis will be able to do the bulk of the work for large applications, even if the cost of some specially complex predicates may still need to be given by the user. In particular, for the examples in Table 1 all results where obtained automatically. Finally, we expect the applications of our analysis to be rather interesting, including resource consumption verification and debugging (including for mobile code), resource control in parallel/distributed computing, and resource-oriented specialization [6,18].

References

1. Aspinall, D., Beringer, L., Hofmann, M., Loidl, H.-W., Momigliano, A.: A program logic for resource verification. In: Slind, K., Bunker, A., Gopalakrishnan, G.C. (eds.) TPHOLs 2004. LNCS, vol. 3223, pp. 34–49. Springer, Heidelberg (2004)
2. Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E., Zolo, T.: Purrs: The Parma University's Recurrence Relation Solver, `http://www.cs.unipr.it/purrs/`
3. Basin, D., Ganzinger, H.: Complexity Analysis based on Ordered Resolution. In: 11th. IEEE Symposium on Logic in Computer Science, IEEE Computer Society Press, Los Alamitos (1996)
4. Bate, I., Bernat, G., Puschner, P.: Java virtual-machine support for portable worst-case execution-time analysis. In: 5th IEEE Int'l. Symp. on Object-oriented Real-time Distributed Computing (April 2002)
5. Chander, A., Espinosa, D., Islam, N., Lee, P., Necula, G.C.: Enforcing resource bounds via static verification of dynamic checks. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 311–325. Springer, Heidelberg (2005)
6. Craig, S.J., Leuschel, M.: Self-tuning resource aware specialisation for Prolog. In: Proc. of PPDP'05, pp. 23–34. ACM Press, New York (2005)
7. Debray, S.K., Lin, N.W.: Cost analysis of logic programs. TOPLAS, 15(5) (1993)
8. Debray, S.K., Lin, N.-W., Hermenegildo, M.: Task Granularity Analysis in Logic Programs. In: Proc. PLDI'90, pp. 174–188. ACM, New York (1990)
9. Debray, S.K., López-García, P., Hermenegildo, M., Lin, N.-W.: Lower Bound Cost Estimation for Logic Programs. In: Proc. ILPS'97, MIT Press, Cambridge (1997)
10. Eisinger, J., Polian, I., Becker, B., Metzner, A., Thesing, S., Wilhelm, R.: Automatic identification of timing anomalies for cycle-accurate worst-case execution time analysis. In: Proc. of DDECS, IEEE Computer Society Press, Los Alamitos (2006)
11. Grobauer, B.: Cost recurrences for DML programs. In: Int'l. Conf. on Functional Programming, pp. 253–264 (2001)

12. Hermenegildo, M., Puebla, G., Bueno, F., López García., P.: Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). Science of Computer Programming 58(1-2), 115–140 (2005)
13. Igarashi, A., Kobayashi, N.: Resource usage analysis. In: Symposium on Principles of Programming Languages, pp. 331–342 (2002)
14. López-García, P., Bueno, F., Hermenegildo, M.: Determinacy Analysis for Logic Programs Using Mode and Type Information. In: Bruynooghe, M. (ed.) Logic Based Program Synthesis and Transformation. LNCS, vol. 3018, pp. 19–35. Springer, Heidelberg (2004)
15. McAllester, D.A.: On the complexity analysis of static analyses. In: Static Analysis Symp., pp. 312–329 (1999)
16. Le Metayer, D.: ACE: An Automatic Complexity Evaluator. ACM Transactions on Programming Languages and Systems 10(2), 248–266 (1988)
17. Nielson, F., Nielson, H.R., Seidl, H.: Automatic complexity analysis. In: European Symposium on Programming, pp. 243–261 (2002)
18. Puebla, G., Ochoa, C.: Poly-Controlled Partial Evaluation. In: Proc. of PPDP'06, pp. 261–271. ACM Press, New York (2006)
19. Rosendhal, M.: Automatic Complexity Analysis. In: Proc. FPCA, ACM, New York (1989)
20. Thiele, L., Wilhelm, R.: Design for time-predictability. In: Perspectives Workshop: Design of Systems with Predictable Behaviour
21. Vasconcelos, P., Hammond, K.: Inferring Cost Equations for Recursive, Polymorphic and Higher-Order Functional Programs. In: Trinder, P., Michaelson, G.J., Peña, R. (eds.) IFL 2003. LNCS, vol. 3145, Springer, Heidelberg (2004)